Strategies to reduce Design Time in Multimodal/Multilingual Dialog Applications

Luis F. D'Haro, Ricardo de Córdoba, Rubén San-Segundo, Juan M. Montero, Javier Macías-Guarasa, José M. Pardo

Speech Technology Group. Dept. of Electronic Engineering. Universidad Politécnica de Madrid
E.T.S.I. Telecomunicación. Ciudad Universitaria s/n, 28040 Madrid, Spain
{lfdharo, cordoba, lapiz, juancho, macias, pardo}@die.upm.es
http://www-gth.die.upm.es

ABSTRACT

In this paper, we present a complete platform for the semiautomatic generation of human-machine dialog systems, that using as input a description of the database of the service, a flow model with the different states of the final application and a guided interaction step by step with the designer's intervention, generates dialogs to access the service data in different languages and two modalities, speech and web, simultaneously. We describe in detail several strategies that have been followed to reduce the time needed to do the design using the mentioned information. We also address important issues in dialog applications as mixed initiative and overanswering dialogs, confirmation handling and how to provide the user long lists of information.

Keywords: automatic dialog systems, automatic generation, multimodality, multilinguality, speech recognition, XML.

1. Introduction

The Gemini project[1] (Generic Environment for Multilingual Interactive Natural Interfaces), is a two year project that is just finishing with the following partners: Knowledge S.A. (Greece) as coordinator, Patras University – WCL (Greece), TEMIC SDS (Germany), UPM – GTH (Spain), FAW (Germany) and Egnatia Bank (Greece). It exploits the results obtained in the IDAS project ([1][2]) and from real-world use of similar systems, to create a generic platform for the development of user-friendly, natural, high quality, intuitive, platform independent, multilingual and multi-modal interactive interfaces to a wide area of databases employed by information service providers.

The main objective of the project has been the development of a platform able to generate dialog applications in a semi-automatic way that are able to handle several languages and several modalities at the same time. This generation should be done with a minimum of cost and human effort. You can see a more detailed description of the work done in the project in [3] and in another paper submitted to the conference. In this paper, we are going to concentrate in all the strategies followed to automate the design and fulfil the main promise we made for the project: automate dialog design.

The dialog generator is a task oriented based system that allows the creation of fixed subtask initiative dialogs [4], and follows a similar approach to the Agenda [5] system (now RavenClaw [6]) from Carnegie Mellon University; in this sense, the designer can build the service using a hierarchical representation of the task and their subcomponents, providing maintainability and scalability, and where each state is expressed as a form-filling task including also information regarding its constraints and optional slots. Moreover, the platform uses some ideas from the USI (Universal Speech Interface [7]) project that are applied for the generation of names of the automatically generated dialogs; the names are derived from the attributes of the database. Also, the assistant creates a XML-based document that allows an easy edition, parsing and specification of the system information structure. Finally, in order to allow different kinds of confirmation, we have selected four types: none, explicit, implicit and no match, based on the confidence value of the recognition.

First, we will make a rough description of the platform architecture to locate the places where dialog design is made and describe which are the sources that are used to automate the design. Then, we will focus on our dialog design modules and how the applications are modeled.

Throughout this article, we will call "user" to the final client of the system, and "designer" to the person that will build the dialog system.

2. Application Generation Platform

The Application Generation Platform (AGP) is the platform used to generate multi-modal dialog applications. The AGP is an integrated set of assistants and tools. Its open and modular architecture simplifies the adaptability of applications designed with the AGP to different use cases. We had several objectives in mind when building the AGP:

- Minimum effort for the designer.
- Standardization. All models generated by the AGP are described in GDialogXML (GEMINI Dialog XML), which is an object-oriented abstract dialogue modeling language [8]. The output is xHTML scripts for Web applications and VoiceXML 2.0 scripts for voice. So, the designer does not need to know any of these languages.
- All models in the AGP can be saved as libraries.
- Service/application independency.

2.1 AGP architecture

The AGP consists of three layers:

1. Top level. The designer specifies the global aspects related to the application and the data. It has 3 assistants.

Application description assistant: the basic characteristics of the application are defined, as the languages, modalities, libraries that will be used, etc.

[1] This work was partly supported by the European Commission's Information Society Technologies Programme under contract no. IST-2001-32343. The authors are solely responsible for the contents of this publication.

Refer to the GEMINI Project Homepage on www.gemini-project.org for further details.

Data modeling assistant: the database classes and their attributes are specified. The use of libraries and the graphical view simplifies the process. The output is the Data Model.

Data connector modeling assistant: the functions to access the database are defined. The objective is that the application is independent of the specific database.

2. Intermediate level. The dialog is defined in a modality and language independent way. It has 3 modules:

State flow assistant (SFA). Here, the states of the dialog are specified (and which states are called from each state) and what slots of information are asked to the user in each state. To specify the slots, attributes from the Data Model are always offered to speed up the design. It is a very high level definition of the dialog.

Retrieval modeling assistant (RMA). Using the information specified in all previous assistants, the designer describes the dialog details. It is the most complex assistant and where a bigger effort has been made to provide flexibility and easiness to the designer, so it is described in detail in section 3.1.

User modeling assistant. Defines the behavior of the system according to the experience of the user (e.g., novice, expert, default, etc.)

3. Bottom level. The dialog is completed with all aspects that are language and modality dependent. It has the following modules:

Modality extension assistant: the dialog is completed with modality dependent aspects, as confirmation handling in user input and list handling for user output. (See section 3.2).

Language extension assistant: the language is completed with language dependent aspects, as the prompts (including SSML tags) or recorded speech, the vocabularies and grammars, etc.

Dialog model linker: links the results of the previous assistants to generate the final dialog model. There is no interaction with the designer.

Then, the AGP generates automatically the scripts in VoiceXML for voice and xHTML for web applications.

There are two more tools in the architecture: a Language modeling tool, where the language models can be specified in the JSGF format, and the Vocabulary builder, where the vocabularies used in the runtime recognizer are prepared.

3. Dialog modeling

We are going to describe in more detail the assistants more related to dialog modeling and the strategies that we have followed to streamline the definition of a dialog.

3.1 Retrieval modeling assistant (RMA)

This assistant is critical, as it is where the dialog is defined in detail. So, it has to be intuitive and has to automate the design, reducing designer effort. As the result has to be modality- and language-independent, we work at a concept level. In Figure 1 we can see the main window of the assistant, with a banking application and its tree-structured diagram flow. It is the result of reading the Dialog flow assistant (SFA), and all dialogs can be edited and completed just double-clicking. We use different colors to indicate if the dialog has been edited or not, the dialog type, if it is a procedure, etc.

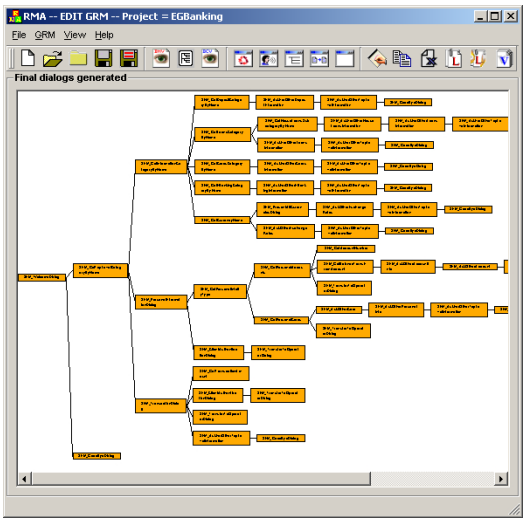

Figure 1. RMA main window.

Now, we will describe in detail which strategies have been applied to speed up dialog design:

1. Several dialogs are automatically created and proposed to the designer which can be drag & dropped anywhere in the assistant: to obtain information from the user (called DGet) and to provide info to the user (called DSay).

DGet and DSay dialogs are based in information from the Data Model (classes and attributes defined for the service). For each class and attribute we generate a DGet and a DSay dialog, which include a tag used by the Language extension assistant to know that the prompt to be presented to the user (for DSay) and the grammar used by the recognizer (for DGet) have to be specified.

We generate several DSay dialogs: a 'DSay class', that lets the selection of specific attributes for the prompt; a 'configurable DSay' for generic prompts; a DSay for each value returned by a database access function; and some predefined DSay's: welcome to the user, goodbye, etc.

Besides the DGet and DSay dialogs, the designer can drag & drop other actions: dialogs from loaded libraries and database access functions. In Figure 2 we can see the auxiliary screen of the RMA, which the designer can use to drag & drop all the dialogs mentioned here.

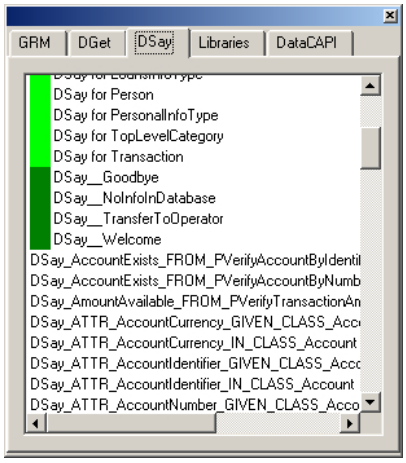

Figure 2. Auxiliary screen of the RMA.

We also provide Dialog Templates to have more complex DGet or DSay dialogs: the designer can create custom dialogs, where we offer the attributes of a particular class and the designer can select those that he needs. Moreover, the object references in the classes are expanded.

2. We make the most of the info from previous assistants.

Besides the Data Model info used to generate automatic dialogs, the main source of the RMA is the output of the State flow assistant (SFA). For each state a dialog is automatically generated. When that dialog is edited, an "SFM proposals" window pops up (see Figure 3), where all info specific to that state is offered to the designer and he does not waste time looking for dialogs in Figure 2 window. It consists of 4 parts:

- Slots asked in the state and next states in the flow.
- State specific DGets: obtained using the slot information (all DGets with a similar name to the slot.)
- Database access functions whose input parameter/s matches any slot of the state.
- State specific DSays: they have as input some value returned by the preceding functions or the slot.

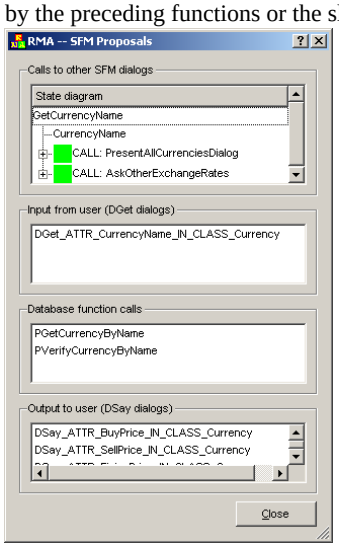

Figure 3. "SFM proposals" window.

In Figure 3 you can see how the dialogs that the designer will need to provide currency information are easily accessible: DGet for CurrencyName to ask the user for the desired currency, followed by PGetCurrencyByName to access the database and get the currency information, some DSays to provide the results of the database access, and finally a call to the next dialog (e.g., AskOtherExchangeRates).

3. The passing of arguments between dialogs is automated.

This is a critical aspect of dialog design. Several dialogs have to be 'connected' as they use the info from the preceding dialog. In typical situations, e.g. to obtain the balance of an account, three dialogs have to be connected. First, we need data from the user (account number). As it belongs to the Data Model, we just need to drag & drop the corresponding DGet. Then, we need the database access function that obtains the account balance from the account number. Again, we just drop it. And finally, we provide the account balance to the user, which is a predefined DSay. In all cases, variable passing is automated, as the correct variable names are proposed to the designer, who just has to Accept them.

4. Dialog types allowed

The user can add several types of dialogs to configure the service. We have considered four basic types: based in a loop, in a sequence of actions (or subdialogs), in information input by the user, and in the value of a variable (a switch construction). As well, we allow empty dialogs, used to specify the call to a dialog that will be defined afterwards; this way, dialogs can be defined following a top-down strategy. Another possibility we offer is dialog cloning, useful when the dialog to be defined is very similar to an existing one.

Besides the possibility to drag & drop the automatic dialogs, several actions can be executed in all these dialogs:

- Define local and global variables
- Insert calls to other dialogs
- Insert if-then-else structures
- Insert switch-case structures
- Insert loops inside the dialog
- Insert assignments for the variables, including a mathematical and strings assistants.

And all these actions can be done without writing anything in our internal syntax [8]. User input is reduced to a minimum.

5. Unified GUI and hotkeys

The GUI used for all dialogs has been unified, offering the possibility to add all mentioned actions in the same way and a recursively (loops – if-then – switches, etc., combined).

6. Mixed initiative and overanswering

The designer can decide if the application will be "system initiative" or "mixed initiative" just selecting a checkbox. For mixed initiative, we need to let the user input two or more slots in the same prompt. We have considered two situations:

- Mixed initiative: the system asks two or more slots in the same prompt.

To define a dialog as mixed initiative, the designer just has to define the slots as such in the SFM, and the RMA will generate automatically a DGet for mixed initiative, which the designer just drops in the relevant window. That is all.

- Overanswering: the system asks one slot but the user answers with additional slots that would have been asked in a later dialog.

For overanswering, whenever the designer drops a DGet he is asked if he wants to define as overanswering slot any other slot defined in the current state or in the following states in the flow. In the real time system, before every DGet the system checks if the slot is already set (as would happen if the user has said it in a previous prompt). If it is set, the system would skip the prompt.

3.2 Modality extension assistant for speech

We take care here of dialog aspects which are specific of a speech application, and so they have to receive a different treatment than in web. The main reason, of course, is that the amount of information that can be provided using speech is much smaller.

3.2.1 Presentation of lists of objects

Lists of objects, which are usually the result of a database query, mean a lot of information. So, they need special

treatment in a speech application. We distinguish four cases as a function of the number of elements of the list.

1. The list is empty.

The system indicates the user that there is no info available and then jumps to a state selected by the designer where he is asked again some slots looking for a less restrictive query.

2. The list has one item.

The designer defines a configurable DSay that provides complete or partial info from the item found.

3. More than one item and less than a maximum allowed.

This is the more complex situation, as the items have to be provided in groups. As the object may have many attributes, we let the designer specify the attributes from each object that have to be played to speed up the process. After playing the info in each group, the user is asked if he wants to continue, repeat the group, begin from scratch, exit or select a specific item to receive more detailed information. When the user selects the item he desires, we play more complete object information.

One situation that we face is when the system finishes reading the whole list and the user does not like any item or has exited before the end of the list. In this case, the system informs the user and repeats the case 1 process.

The designer has also the option to inform the user how many items are there in the list, and the user can choose how many he wants to listen to. This way, he can reduce dynamically the length of the information provided.

4. More items than the maximum allowed.

As there are too many items, the search has to be more restrictive. We can have two different situations. First, if all slots of the application are already filled, the user has to change some of them to make them more restrictive (e.g., he wants to fly next week but there are too many flights); the designer specifies those slots and the questions are repeated. In the second case, if there are still some slots to be asked the system continues with the normal dialog flow until the next database query.

We have also considered a simplified case: when the list depends only on one slot input by the user. In this case, we present a simplified version of the previous windows where the designer does not need to specify the slots to be unset.

All the values that determine the behavior of list presentation, as the maximum number of items allowed, are user-level dependent, so they are assigned dynamically in the real-time system according to the User Model.

3.2.2 Confirmation handling

The result of speech recognition has to be confirmed before making a database query. We confirm both normal slots and overanswering slots. We have considered two types of confirmation in our assistant: Simple and Complete. Simple is recommended for dialogs with a very high confidence, as Yes/No or passwords questions. Complete uses several levels of confidence to determine the confirmation type: none, implicit, explicit or repeat the question (like in a no match situation).

The assistant selects automatically the input dialogs (DGet), decides which should be Simple and which Complete (the designer can change it) and generates the flow to do all the confirmation types, all in a transparent way to the designer. Internally, the assistant also generates confirmation dialogs that are used to make implicit/explicit confirmations.

The assistant also controls where implicit confirmation should not be allowed. For example, if the next step in dialog flow is the database access, explicit confirmation should be used regardless of the confidence level. Moreover, the assistant determines automatically where to jump to in case the user rejects the implicit confirmation (the rejection is detected in the following state in the flow).

4. CONCLUSIONS

We have developed a dialog generation platform which is both powerful and flexible, and at the same time provides a high degree of automation. It is able to generate in a semi-automatic way dialogs valid for multiple languages and two modalities using just a description of the database, a basic state flow of the application and a simplified user-friendly interaction with the designer.

We have shown several strategies to speed up the design process using automatic dialogs, automating the passing of arguments and allowing the definition of any dialog or construct. We have also included the possibility to handle mixed initiative and overanswering dialogs using the same framework. Detailed procedures to handle list and confirmation handling have been presented too.

The result is an open and portable platform that can be used to generate quickly dialog applications.

5. REFERENCES

[1] Lehtinen, G., S. Safra, …, J.M. Pardo, R. Córdoba, R. San-Segundo, et al. “IDAS: Interactive Directory Assistance Service”, VOTS-2000 Workshop, Belgium.

[2] R. Córdoba, et al. “An Interactive Directory Assistance Service for Spanish with Large-Vocabulary Recognition”, Eurospeech 2001, pp. 1279-1282.

[3] Hamerich, S. W., V. Schubert, V. Schless, R. Córdoba, J.M. Pardo, L.F. d’Haro, B. Kladis, O. Kocsis, S. Igel. “Semi-Automatic Generation of Dialogue Applications in the GEMINI Project”, Sigdial 2004.

[4] Hearst M. A, Allen J, Horvitz, E, Guinn C. 1999. “Mixed-initiative interaction”. IEEE Intelligent Systems, Vol 14(5): pp.14-23.

[5] Rudnicky, A. and Xu W. “An agenda-based dialog management architecture for spoken language systems”. IEEE ASRU Workshop 1999: pp I-337-340.

[6] Bohus, D. and Rudnicky, A. “RavenClaw: Dialogue Management Using Hierarchical Task Decomposition and an Expectation Agenda”. Eurospech 2003: pp 597-600.

[7] Toth, A.R. T. K. Harris, et al. “Towards every-citizen’s speech interfaces: an application generator for speech interfaces to databases”. ICSLP 2002, pp. 1497–1500.

[8] Hamerich, S.W., et al. “XML-Based Dialogue Descriptions in the GEMINI Project”. Proceedings of the “Berliner XML-Tage 2003”, Germany, pp. 404-412.